Mahidol University
Wisdom of the Land

Faculty of Information and Communication Technology

LECTURE 08

MIDTERM REVIEW

ITCS208 Object Oriented Programming

Dr. Siripen Pongpaichet

Slide Credit:

Dr. Petch Sajjacholapunt & Dr. Suppawong Tuarob & ITCS 208 [2/2015]

Modified from Cay Horstmann's

12 March 2020

Semester 2/2020

Course Learning Outcome

CLO1 : Explain the concepts of Object-Oriented Programming as well as the purpose and usage principles of encapsulation, polymorphism, and inheritance

CLO2 : Identify structures and outputs of a given source code written in Object-Oriented Programming paradigm

CLO3 : Design classes, objects, members of a class (e.g. attributes, methods, and data types) and the relationships among them needed for a specific problem

CLO4 : Develop application programs that appropriately use Object-Oriented Programming concepts and practices (e.g. classes, interfaces, access control identifiers, and error exception handling) to solve a given problem

CLO5 : Implement Object-Oriented Programs to solve common computer science problems (e.g. recursion, sorting, and searching)

What have we learned so far...

OOP Paradigm: Object and Class

Data Types

Decision (If-Else, Switch)

Iteration (For, Do, While Loop)

Array

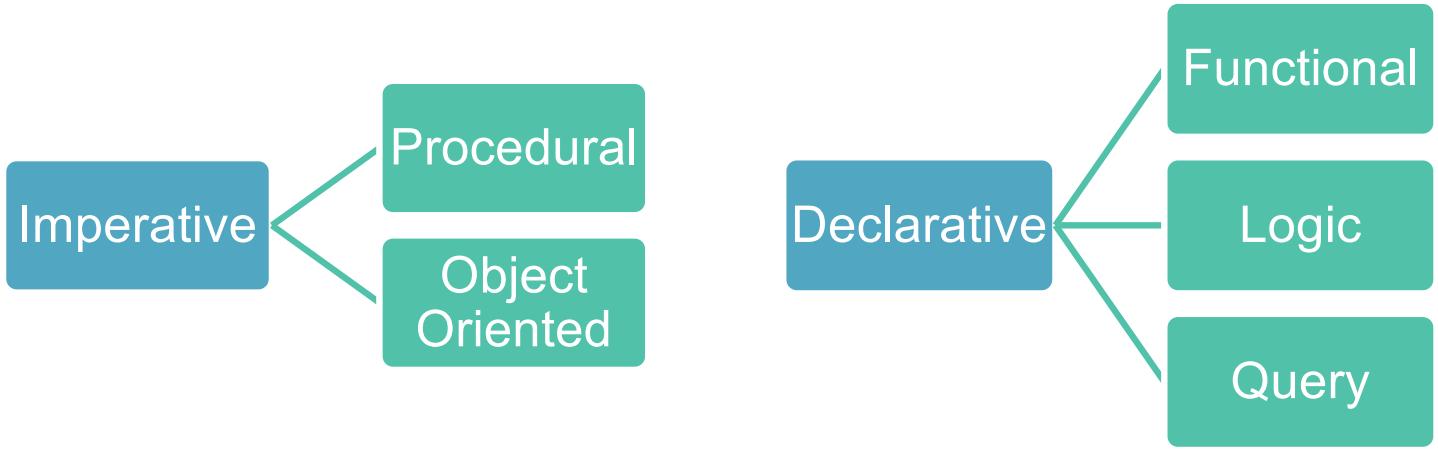
ArrayList

Inheritance

Polymorphism

What is a Programming Paradigm?

It is a philosophy, style, or general approach to writing a code



It is not about a specific programming language

- One language can be used or applied in many paradigms

PROCEDURAL

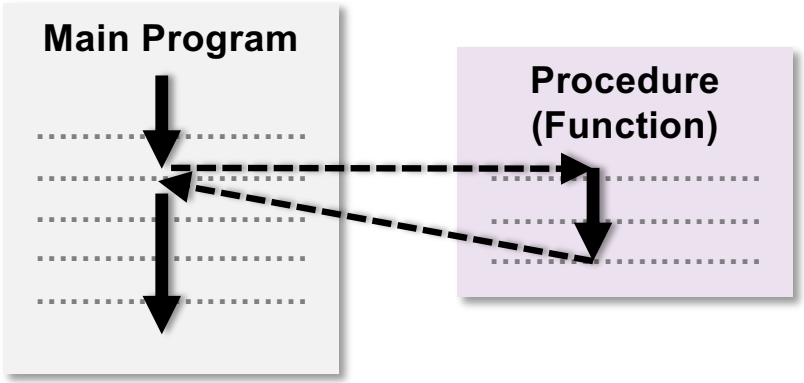
Focuses on writing good *functions* and *procedures*

Describes computation in term of a *sequence of instructions* that change the program state

Top-Down Design

- Try to break the problem to be sub-problems

It can be difficult to maintain.



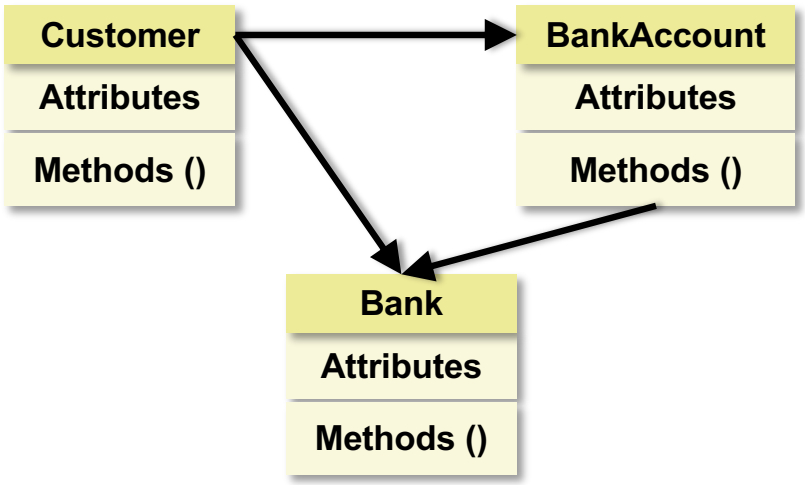
OBJECT-ORIENTED

Uses *abstraction* to create models based on the real world environment

Describes computation in term of objects that perform certain actions (methods)

Object-Oriented Design

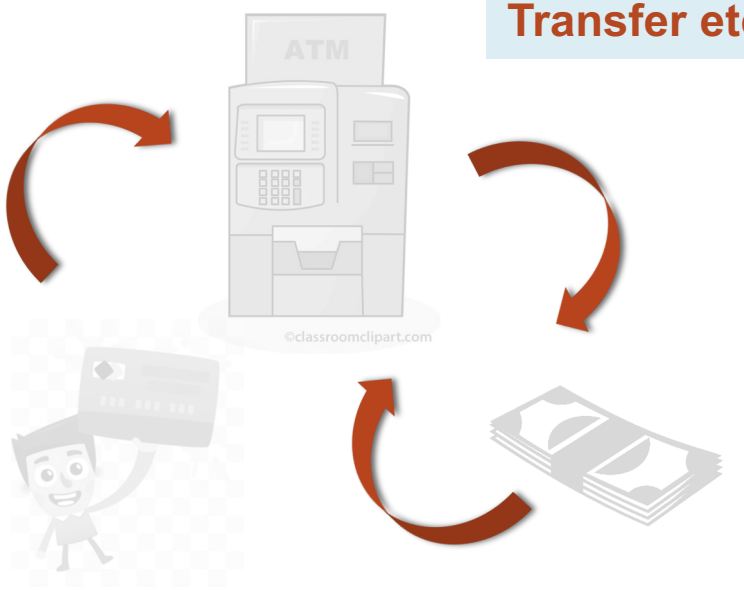
- In form of classes and objects



Real-world Example: ATM Machine

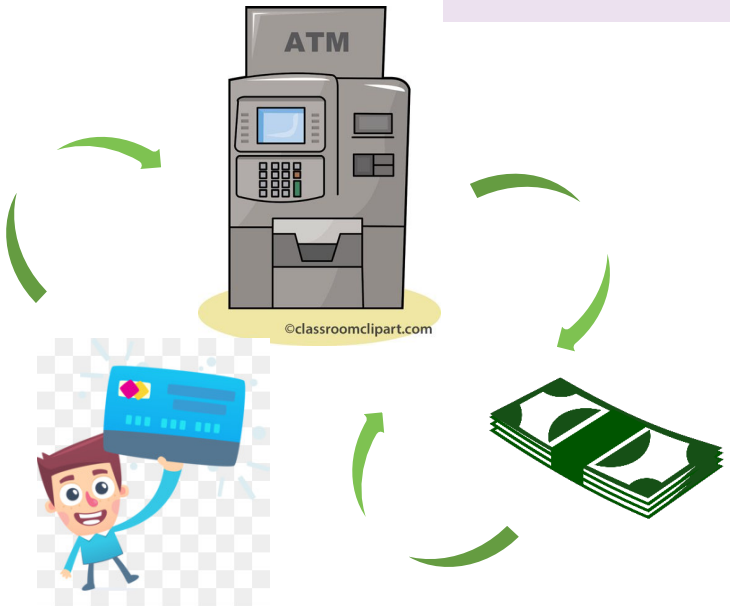
Procedural

Authenticate
Withdraw
Deposit
Transfer etc.



Object-Oriented

Customer
Bank Account
Debit Card
ATM machine

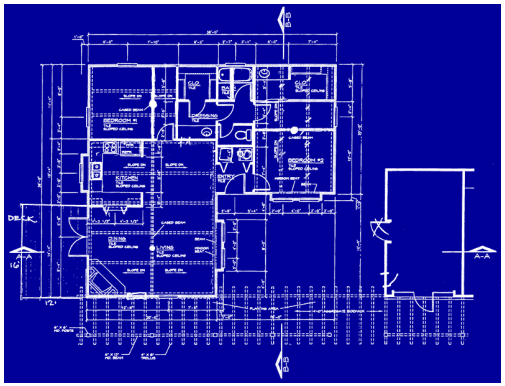


More Examples: Speaker

Speaker blueprint

State:
- current song
- volume
- battery life
- color

Behavior:
- power on/off
- change song
- change volume



creates

Speaker #1

state:
- song = "1,000,000 Miles"
- volume = 17
- battery life = 2.5 hrs
- color = "red"

behavior:
- power on/off
- change song
- change volume

Speaker #2

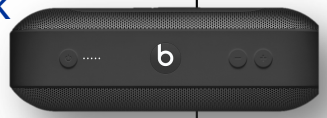
state:
- song = "Letting You"
- volume = 9
- battery life = 3.41 hrs
- color = "white"

behavior:
- power on/off
- change song
- change volume

Speaker #3

state:
- song = "Discipline"
- volume = 24
- battery life = 1.8 hrs
- color = "black"

behavior:
- power on/off
- change song
- change volume



More Examples: Car

Car

- brand: String
- model: String
+ price: double

+ Car(carBrand: String, carModel: String, carPrice: double)
+ setModel(carModel: String)
+ getModel(): String
+ updateBrand(carBrand: String, carModel: String)
+ toString(): String

More Examples: Products in the Vending Machine

Product
- name: String - price: double - quantity: int - discount: double
+ Product (name: String, price: double, qty: int) + setDiscount (d: double) + refill (qty: int) + isSoldout (): boolean + toString (): String

state

behavior

Object-Oriented Designed for 'Course Enrollment'

Object

Student2
name: Lisa id: 1235 email: 1235@xyz.edu
registerCourse(…) dropCourse(…) getGPA()

CLASS

Student	Course	Instructor
Attributes		
Methods		

Object-Oriented Designed for 'Course Enrollment'

Object

Student1
name: Phillip id: 1234 email: 1234@xyz.edu …
registerCourse(…) dropCourse(…) getGPA() …

Attributes

Methods

Student2
name: Lisa id: 1235 email: 1235@xyz.edu …
registerCourse(…) dropCourse(…) getGPA() …

Course1
name: OOP id: MTS280 Instructor:Teacher1 …
setInstructor(…) addStudent(…) removeStudent(…) getTotalStudents() …

Teacher1
name: Ing id: 71 email: 71@xyz.edu …
teachCourse(…) addCourseGrade(…) …

Implementing Class

Car.java

```
public class Car {
```

Attributes/Fields

```
    private String brand;
    private String model;
    public double price;
```

Constructor

```
    public Car(String carBrand, String carModel, double carPrice){
        brand = carBrand;
        model = carModel;
        price = carPrice;
    }
```

Setter method

```
    public void setModel(String carModel){
        model = carModel;
    }
```

Getter method

```
    public String getModel(){
        return model;
    }
```

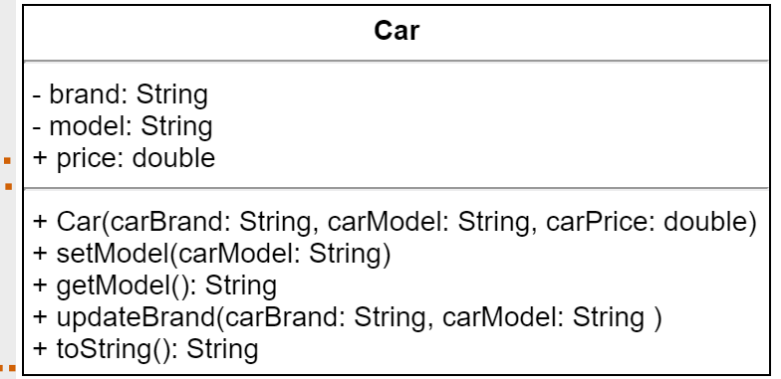
Instance method with no return

```
    public void updateBrand(String carBrand, String carModel){
        brand = carBrand;
        model = carModel;
    }
```

Instance method with return

```
    public String toString(){
        return "Car[brand=" + brand + ", model=" + model + ", price=" + price + "]";
    }
```

```
}
```



Implementing Class Tester

Declaration – variable name and object type
Instantiation – 'new' keyword
Initialization – call to a constructor

```
public class CarTester {

    public static void main(String[] args) {

        Car myCar = new Car("Toyota","Corolla Altis",970000);

        myCar.setModel("Camry");
        System.out.println(myCar.getModel());

        myCar.price = 880900;
        System.out.println(myCar.price);

        myCar.updateBrand("Mercedes-Benz", "C350e");
        myCar.price = 3280900;
        System.out.println(myCar);

    }

}
```

More Examples: `Point` objects

```
import java.awt.*;
...
Point p1 = new Point(5, -2);
Point p2 = new Point();          // origin (0, 0)
```

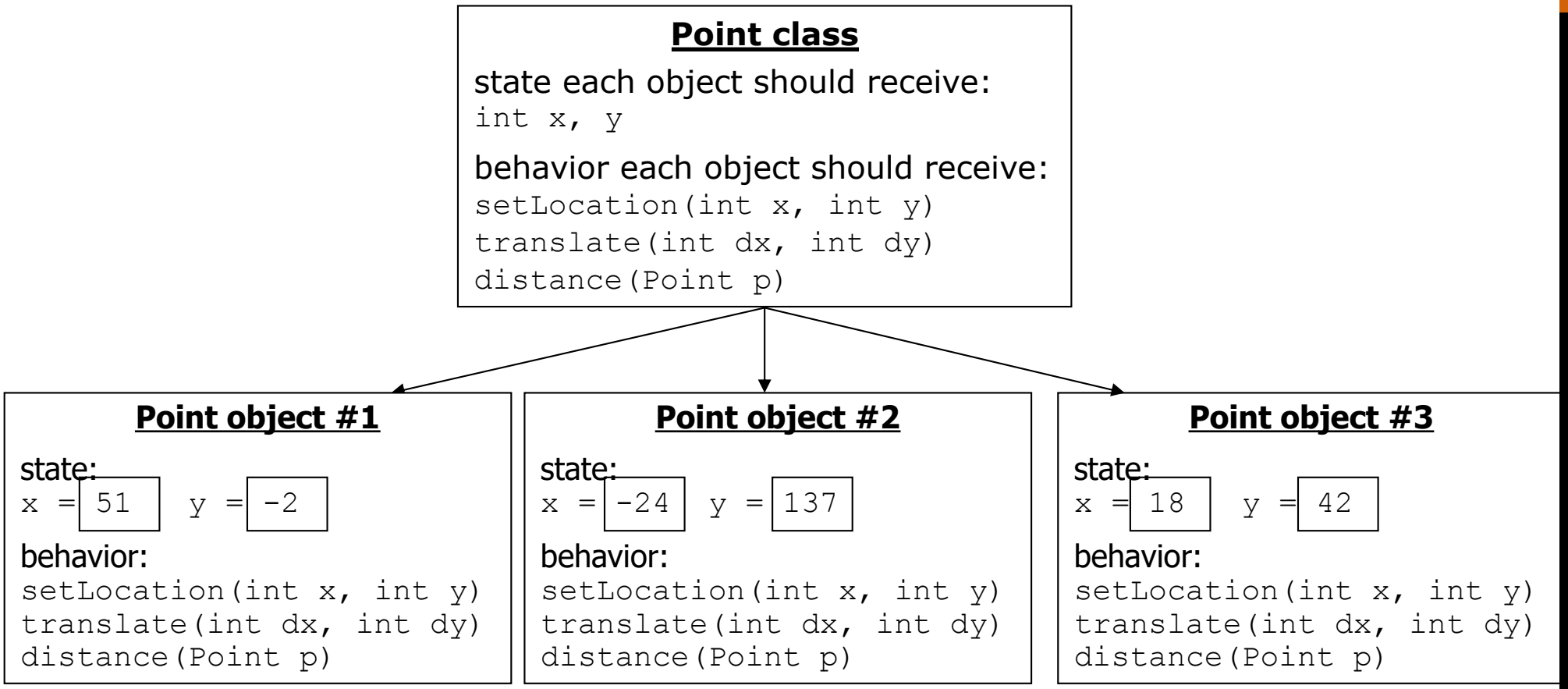
Data:

Name	Description
`x`	the point's x-coordinate
`y`	the point's y-coordinate

Methods:

Name	Description
`setLocation(`**`x, y`**`)`	sets the point's x and y to the given values
`translate(`**`dx, dy`**`)`	adjusts the point's x and y by the given amounts
`distance(`**`p`**`)`	how far away the point is from point *p*

`Point` class as blueprint



- The class (blueprint) describes how to create objects.
- Each object contains its own data and methods.
  - The methods operate on that object's data.

Constructors

Constructor: Initializes the state of new objects.

```
public type(parameters) {
    statements;
}
```

- runs when the client (a main program that uses object) uses the `new` keyword
- no return type is specified; implicitly "returns" the new object

```
public class Point {
    private int x;
    private int y;

    public Point(int initialX, int initialY) {
        x = initialX;
        y = initialY;
    }
```

Multiple constructors

A class can have multiple constructors.

- Each one must accept a unique set of parameters.

Example: **A `Point` constructor with no parameters that initializes the point to (0, 0).**

```
public class Point {
        private int x;
        private int y;

        public Point(int initialX, int initialY) {
            x = initialX;
            y = initialY;
        }
        // Constructs a new point at (0, 0).
        public Point() {
            x = 0;
            y = 0;
        }
```

Default Constructor

The keyword `this`

`this` : Refers to the implicit parameter inside your class.

(a variable that stores the object on which a method is called)

- Refer to a field: `this.`**field**
- Call a method: `this.`**method**`(`**parameters**`);`
- One constructor `this(`**parameters**`);` can call another:

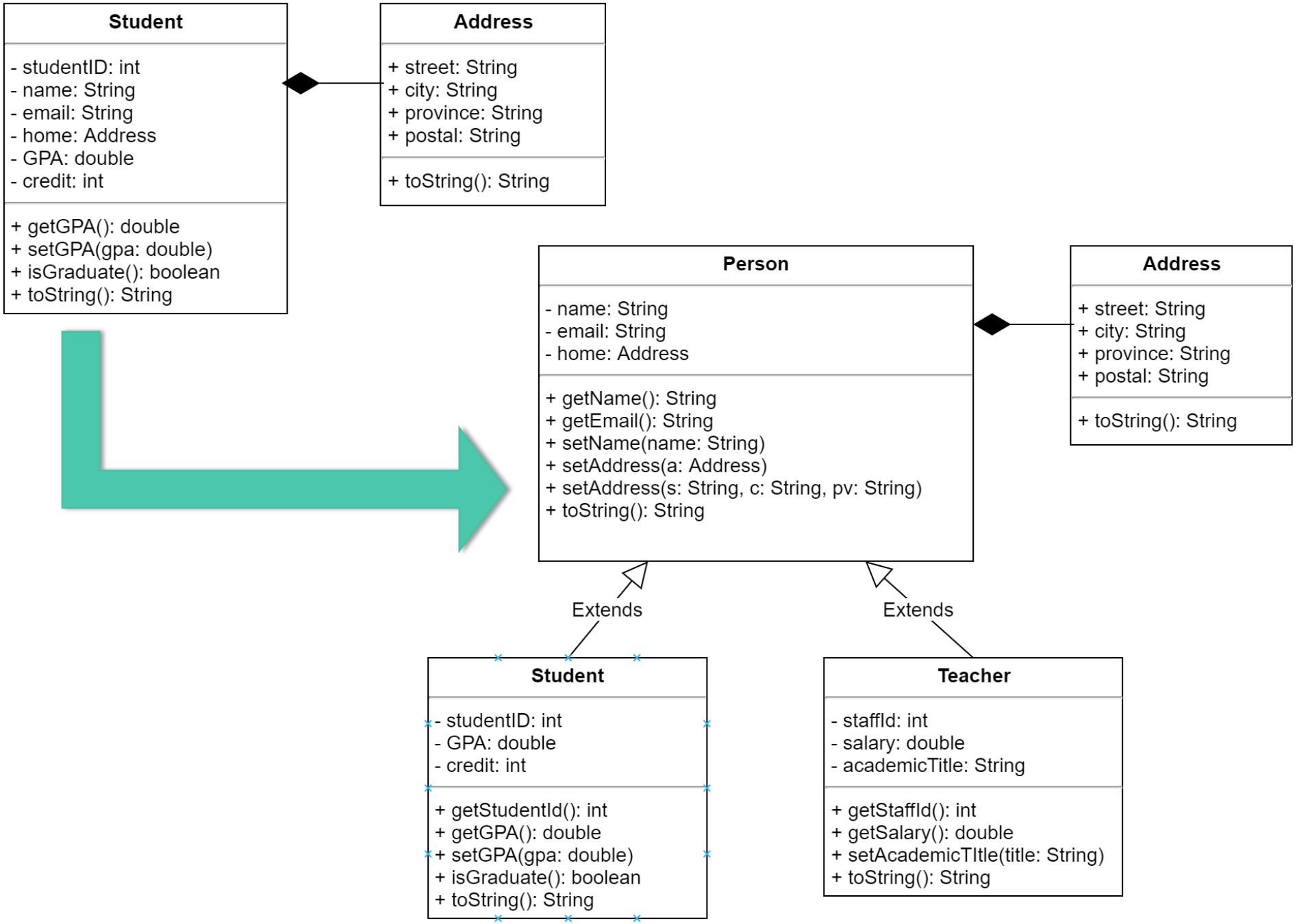
Calling another constructor

```
public class Point {
    private int x;
    private int y;

    public Point() {
        this(0, 0);
    }

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    ...
}
```

Overload Constructor

- Avoids redundancy between constructors
- Only a constructor (not a method) can call another constructor

Advanced Design with Inheritance



Primitives vs. objects

In java, a variable can be either

- Primitive
  - Has well-defined set of possible values
  - Primitive types are lowercase and shown as `keyword` in Eclipse.
  - E.g. int, double, char, boolean
- Reference
  - Similar to C/C++, You can think of it as a pointer to an object.
  - An **object** is a class instance or an **array**.
  - Yes, an array is an object!

A swap method?

Does the following `swap` method work? Why or why not?

```
public static void main(String[] args) {
    int a = 7;
    int b = 35;

    // swap a with b?
    swap(a, b);

    System.out.println(a + " " + b);
}

public static void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```

https://codebunk.com/b/317307115/

Primitive: Value semantic

value semantics: Behavior where values are copied when assigned, passed as parameters, or returned.

- All primitive types in Java use value semantics.
- When one variable is assigned to another, its value is copied.
- Modifying the value of one variable does not affect others.

```
int x = 5;
int y = x;      // x = 5, y = 5
y = 17;         // x = 5, y = 17
x = 8;          // x = 8, y = 17
```

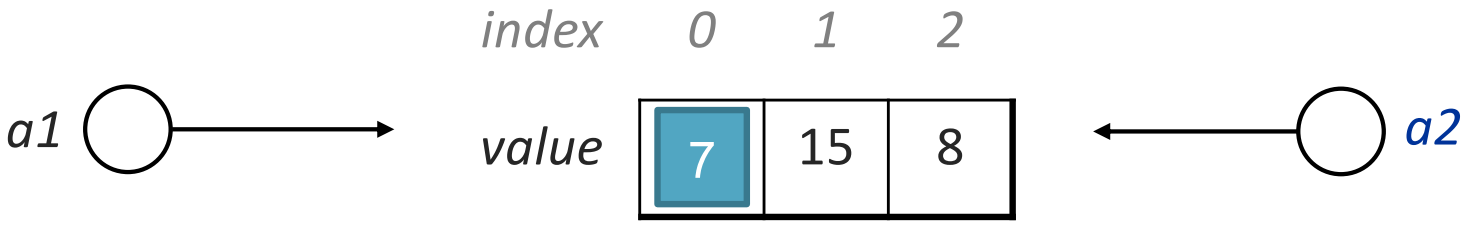
Array: Reference semantic

reference semantics: Behavior where variables actually store the address of an object in memory.

- When one variable is assigned to another, the object is *not* copied; both variables refer to the *same object*.
- Modifying the value of one variable *will* affect others.

```
int[] a1 = {4, 15, 8};
int[] a2 = a1;     // refer to same array as a1
a2[0] = 7;
System.out.println(Arrays.toString(a1));
```

?

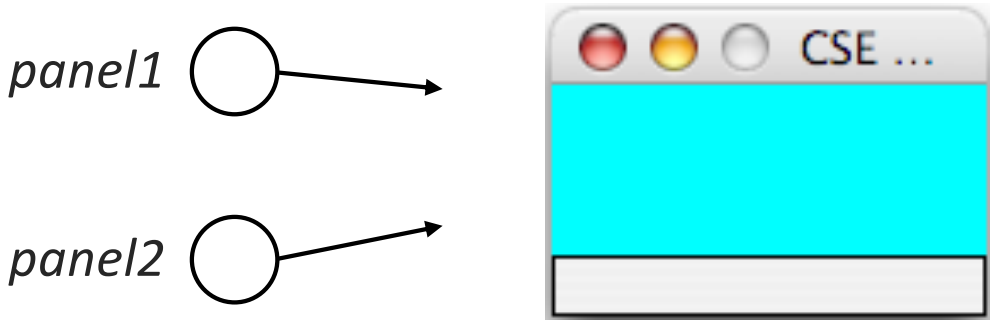


Object: Reference semantic

Arrays and objects use reference semantics. Why?

- *efficiency.* Copying large objects slows down a program.
- *sharing.* It's useful to share an object's data among methods.

```
DrawingPanel panel1 = new DrawingPanel(80, 50);
DrawingPanel panel2 = panel1;   // same window
panel2.setBackground(Color.CYAN);
```

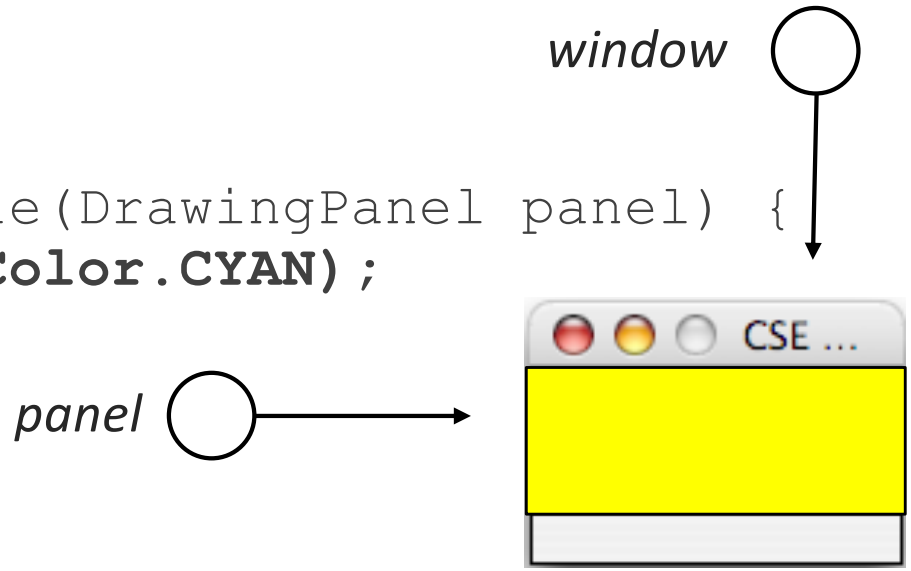


Objects as parameters

When an object is passed as a parameter, the object is *not* copied. The parameter refers to the same object.

- If the parameter is modified, it *will* affect the original object.

```
public static void main(String[] args) {
    DrawingPanel window = new DrawingPanel(80, 50);
    window.setBackground(Color.YELLOW);
    example(window);
}

public static void example(DrawingPanel panel) {
    panel.setBackground(Color.CYAN);
    ...
}
```

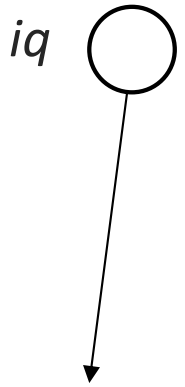


Arrays pass by reference

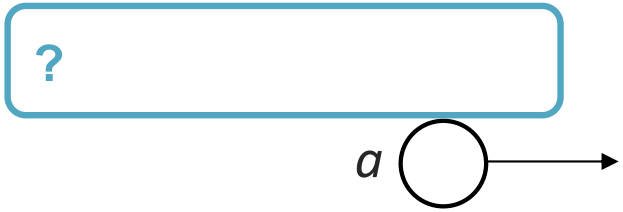
Arrays are also passed as parameters by reference.

- Changes made in the method are also seen by the caller.

```
public static void main(String[] args) {
    int[] iq = {126, 167, 95};
    increase(iq);
    System.out.println(Arrays.toString(iq));
}

public static void increase(int[] a) {
    for (int i = 0; i < a.length; i++) {
        a[i] = a[i] * 2;
    }
}
```



- Output:



index	0	1	2
value	252	334	190

```java
int[] numArr = new int[5];
numArr[0] = 2;
numArr[2] = 5;              //numArr[5] = 10; //ERROR

System.out.println("----------- Array --------------");
for(int i = 0; i < numArr.length; i++){
    System.out.println("value at index " + i + " is " + numArr[i]);
}
```

```
---------- Array -------------
value at index 0 is 2
value at index 1 is 0
value at index 2 is 5
value at index 3 is 0
value at index 4 is 0
```

```java
List<String> nameList = new ArrayList<String>();
nameList.add("a");
nameList.add("b");

System.out.println("----------- List --------------");
for(int i = 0; i < nameList.size(); i++){
    System.out.println("name at index "+i+" is "+ nameList.get(i));
}
```

```
---------- List -------------
name at index 0 is a
name at index 1 is b
```

```java
class Card{
    String name;
    int hp;

    public Card(String name, int hp){
        this.name = name;
        this.hp = hp;
    }

    public int getHP(){
        return this.hp;
    }

    public void setHP(int hp){
        this.hp = hp;
    }

    public String toString(){
        return "name: " + this.name + ", hp: " + this.hp;
    }
}
```

```java
Card[] cardArray = new Card[5];
cardArray[0] = new Card("Mario", 5000);
cardArray[2] = new Card("Pikachu", 7000);

System.out.println("---------- Array -------------");
for(int i = 0; i < cardArray.length; i++){
    System.out.println("value at index " + i + " is " + cardArray[i]);
}
```

```
---------- Array -------------
value at index 0 is name: Mario, hp: 5000
value at index 1 is null
value at index 2 is name: Pikachu, hp: 7000
value at index 3 is null
value at index 4 is null
```

```java
List<Card> cardList = new ArrayList<Card>();
cardList.add(new Card("Doraemon", 10000));
cardList.add(new Card("Nobita", 200));

System.out.println("---------- List -------------");
for(int i = 0; i < cardList.size(); i++){
    System.out.println("name at index " + i + " is " + cardList.get(i));
}
```

```
---------- List -------------
name at index 0 is name: Doraemon, hp: 10000
name at index 1 is name: Nobita, hp: 200
```

```java
int[] numArr = new int[5];
numArr[0] = 2;
numArr[2] = 5;                    //numArr[5] = 10; //ERROR

System.out.println("---------- Array -------------");
for(int i = 0; i < numArr.length; i++){
     System.out.println("value at index " + i + " is " + numArr[i]);
}
```

```
---------- Array -------------
value at index 0 is 2
value at index 1 is 0
value at index 2 is 5
value at index 3 is 0
value at index 4 is 0
```

```java
List<String> nameList = new ArrayList<String>();
nameList.add("a");
nameList.add("b");

System.out.println("---------- List -------------");
for(int i = 0; i < nameList.size(); i++){
     System.out.println("name at index " + i + " is " + nameList.get(i));
}
```

```
---------- List -------------
name at index 0 is a
name at index 1 is b
```

```java
Card[] cardArray = new Card[5];
cardArray[0] = new Card("Mario", 5000);
cardArray[2] = new Card("Pikachu", 7000);

System.out.println("---------- Array -------------");
for(int i = 0; i < cardArray.length; i++){
     System.out.println("value at index " + i + " is " + cardArray[i]);
}
```

```
---------- Array -------------
value at index 0 is name: Mario, hp: 5000
value at index 1 is null
value at index 2 is name: Pikachu, hp: 7000
value at index 3 is null
value at index 4 is null
```

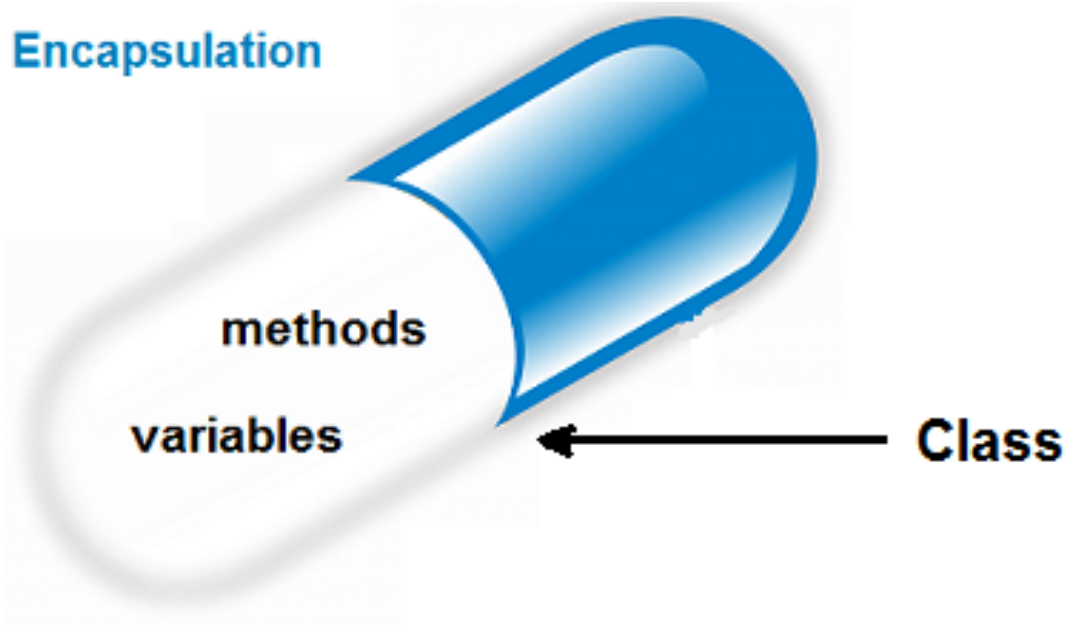
```java
List<Card> cardList = new ArrayList<Card>();
cardList.add(new Card("Doraemon", 10000));
cardList.add(new Card("Nobita", 200));

System.out.println("---------- List -------------");
for(int i = 0; i < cardList.size(); i++){
     System.out.println("name at index " + i + " is " + cardList.get(i));
}
```

```
---------- List -------------
name at index 0 is name: Doraemon, hp: 10000
name at index 1 is name: Nobita, hp: 200
```

Encapsulation

Hiding implementation details from clients (main program).

- Encapsulation enforces *abstraction*.
  - separates external view (behavior) from internal view (state)
  - protects the integrity of an object's data



Benefits of Encapsulation

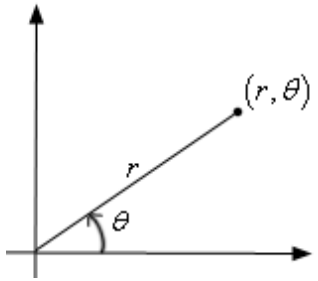
Abstraction between object and clients (main program)

Protects object from unwanted access

- Example: Can't fraudulently increase an `Account`'s balance.

Can change the class implementation later

- Example: `Point` could be rewritten in polar coordinates (r, θ) with the same methods.



Can constrain objects' state (invariants)

- Example: Only allow `Account`s with non-negative balance.
- Example: Only allow `Date`s with a month from 1-12.

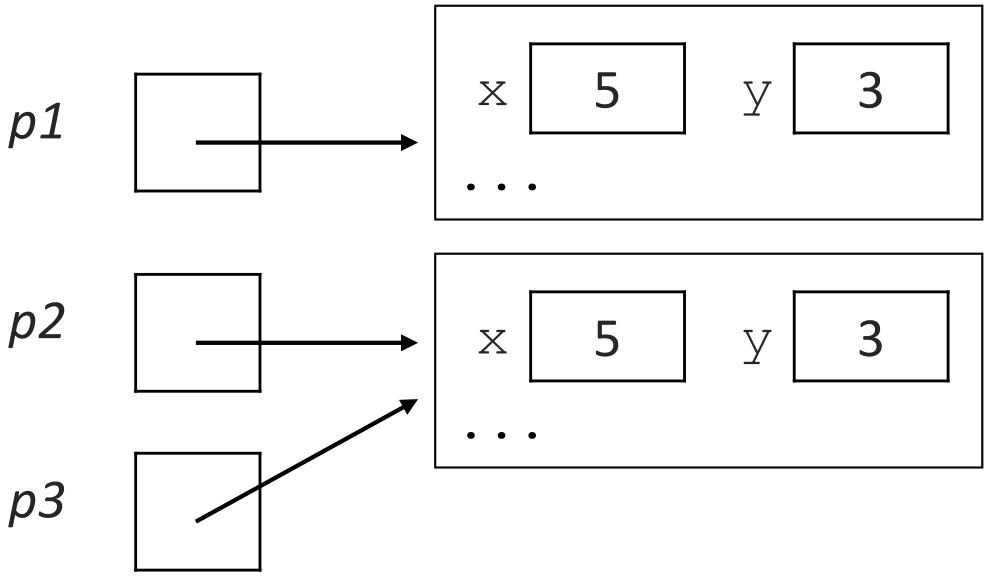
Comparing objects

The == operator does not work well with objects.

`==` compares references to objects, not their state.

It only produces `true` when you compare an object to itself.

```
Point p1 = new Point(5, 3);
Point p2 = new Point(5, 3);
Point p3 = p2;

// p1 == p2 is false;
// p1 == p3 is false;
// p2 == p3 is true
```



The `equals` method

The `equals` method compares the state of objects.

```
if (str1.equals(str2)) {
    System.out.println("the strings are equal");
}
```

When writing a class, its default `equals` method behaves like `==`

```
if (p1.equals(p2)) {    // false :-(
    System.out.println("equal");
}
```

- This is the default behavior we receive from class `Object`.
- Java doesn't understand how to compare new classes by default.

Override `equals` method

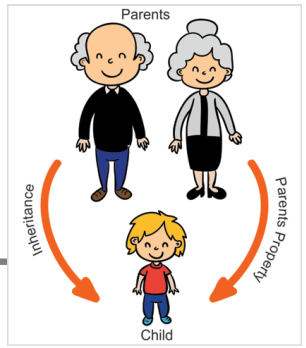
```
class Point {
    private int x;
    private int y;

    public Point(int initialX, int initialY) {
        x = initialX;
        y = initialY;
    }

    public int getX(){ return x;}
    public int getY(){ return y;}

    public boolean equals(Point p){
        if(this.x == p.getX() && this.y == p.getY())
            return true;
        else
            return false;
    }
}
```

Challenge:
Can you make this `equals` method shorter?

```
Point p1 = new Point(5, 3);
Point p2 = new Point(5, 3);
Point p3 = p2;
System.out.println(p1.equals(p2));
System.out.println(p2.equals(p3));
```

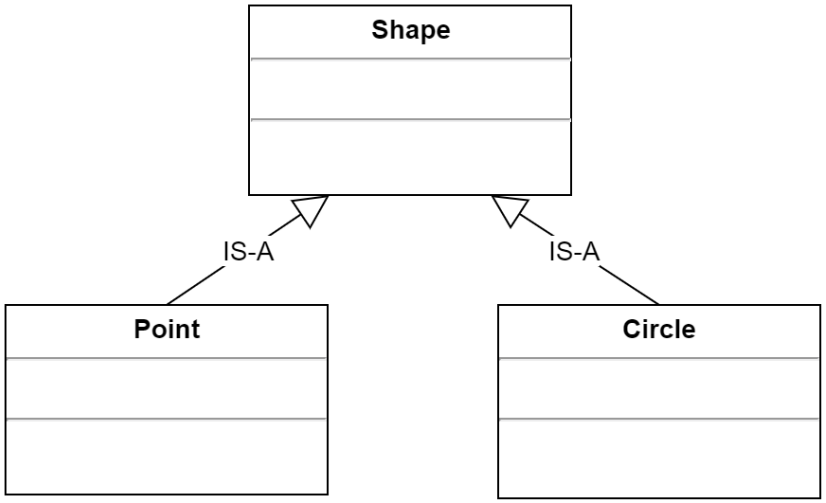
?

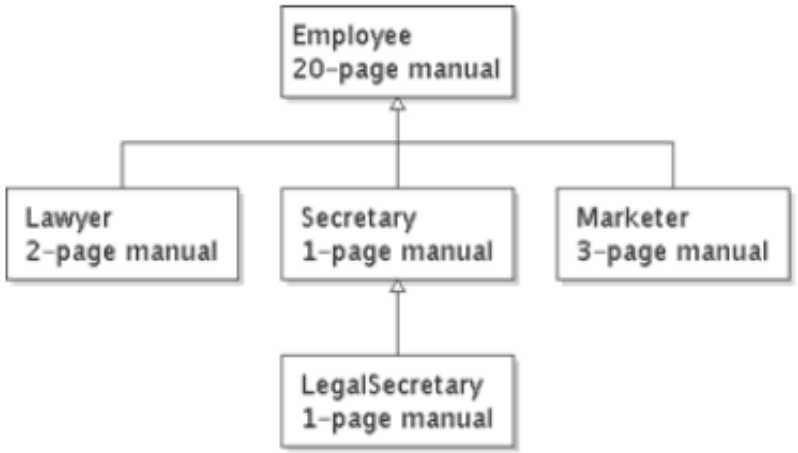
Inheritance



Forming new classes based on existing ones.

- a way to share/**reuse code** between two or more classes
- **superclass**: Parent class being extended.
- **subclass**: Child class that inherits behavior from superclass.
  - gets a copy of every field and method from superclass
- **is-a relationship**:
  Each object of the subclass also "is a(n)" object of the superclass and can be treated as one.





Inheritance syntax

```
public class name extends superclass {
```

- Example:

```
public class Lawyer extends Employee {
    ...
}
```

By extending `Employee`, each `Lawyer` object now:

- receives a copy of each method from `Employee` automatically
- can be treated as an `Employee` by client code

Lawyer can also replace ("override") behavior from Employee.

Overriding Methods

Override: To write a new version of a method in a subclass that replaces the superclass's version.

- No special syntax required to override a superclass method.
  Just write a new version of it in the subclass.

```
public class Lawyer extends Employee {

// overrides getVacationForm in Employee class
    public String getVacationForm() {
        return "pink";
    }
    ...
}
```

Cloning: b = a.clone();

Shallow copy

- b and a have references to common underlying objects.

Deep copy

- b and a don't have any references to common objects.

```
CharSequence[] str1 = new CharSequence[2];
str1[0] = new String("MU");
str1[1] = new StringBuilder("ICT");

CharSequence[] shallowCopy = new CharSequence[2];
for(int i = 0; i < str1.length; i++)
{
  shallowCopy[i] = str1[i];
}

CharSequence[] deepCopy = new CharSequence[2];
for(int i = 0; i < str1.length; i++)
{
  if(str1[i] instanceof String)
  {
      deepCopy[i] = new String((String)str1[i]);
  }
  else if(str1[i] instanceof StringBuilder)
  {
      deepCopy[i] = new StringBuilder();
     ((StringBuilder)deepCopy[i]).append(str1[i]);
  }
}
```

The `super` keyword

A subclass can call its parent's method/constructor:

```
super.method(parameters)        // method
super(parameters);                    // constructor
```

```
public class Lawyer extends Employee {
     public Lawyer(String name) {
          super(name);
     }

     // give Lawyers a $5K raise (better)
     public double getSalary() {
          double baseSalary = super.getSalary();
          return baseSalary + 5000.00;
     }
}
```

Subclasses and fields

```
public class Employee {
    private double salary;
    ...
}

public class Lawyer extends Employee {
    ...
    public void giveRaise(double amount) {
    salary += amount;   // error; salary is private
    }
}
```

Inherited private fields/methods cannot be directly accessed by subclasses. ***(The subclass has the field, but it can't touch it.)***

- How can we allow a subclass to access/modify these fields?

Inheritance and constructors

If we add a constructor to the `Employee` class, our subclasses do not compile. The error:

```
Lawyer.java:2: cannot find symbol
symbol  : constructor Employee()
location: class Employee
public class Lawyer extends Employee {
       ^
```

- The short explanation: Once we write a constructor (that requires parameters) in the superclass, we must now write constructors for our employee subclasses as well.

Inheritance and constructors

Constructors are not inherited.

- Subclasses don't inherit the `Employee(int)` constructor.
- Subclasses receive a default constructor that contains:

```
public Lawyer() {
    super();          // calls Employee() constructor
}
```

But our `Employee(int)` replaces the default `Employee()`.

- The subclasses' default constructors are now trying to call a non-existent default `Employee` constructor.

Calling superclass constructor

`super(`**parameters**`);`

- Example:

```
public class Lawyer extends Employee {
    public Lawyer(int years) {
        super(years);  // calls Employee constructor
    }
    ...
}
```

- The `super` call must be the first statement in the constructor.

Polymorphism

Ability for the same code to be used with different types of objects and behave differently with each.

- `System.out.println` can print any type of object. **(Overloading)**
  - Each one displays in its own way on the console.

- `Human` can command different animal to `speak()`. **(Overriding)**

Coding with polymorphism

A variable of type *T* can hold an object of any subclass of *T*.

```
Employee ed = new Lawyer();
```

- You can call any methods from the `Employee` class on `ed`.

When a method is called on `ed`, it behaves as a `Lawyer`.

```
System.out.println(ed.getSalary());   // 50000.0
System.out.println(ed.getVacationForm()); // pink
```

Polymorphic parameters

You can pass any subtype of a parameter's type.

```
public static void main(String[] args) {
    Lawyer lisa = new Lawyer();
    Secretary steve = new Secretary();
    printInfo(lisa);
    printInfo(steve);
}
```

```
public static void printInfo(Employee e) {
  System.out.println("pay   : " + e.getSalary());
  System.out.println("vdays: " + e.getVacationDays());
  System.out.println("vform: " + e.getVacationForm());
  System.out.println();
}
```

OUTPUT:

```
pay   : 50000.0    pay   : 50000.0
vdays: 15          vdays: 10
vform: pink        vform: yellow
```

Polymorphism and arrays

Arrays of superclass types can store any subtype as elements.

```
public static void main(String[] args) {
    Employee[] e = {new Lawyer(),   new Secretary(),
                    new Marketer(), new LegalSecretary()};

for (int i = 0; i < e.length; i++) {
System.out.println("pay  : " + e[i].getSalary());
System.out.println("vdays: " + i].getVacationDays());
        System.out.println();
  }
}
```

Output:

```
pay  : 50000.0      pay  : 60000.0
vdays: 15           vdays: 10

pay  : 50000.0      pay  : 55000.0
vdays: 10           vdays: 10
```

Casting references

A variable can only call that type's methods, not a subtype's.

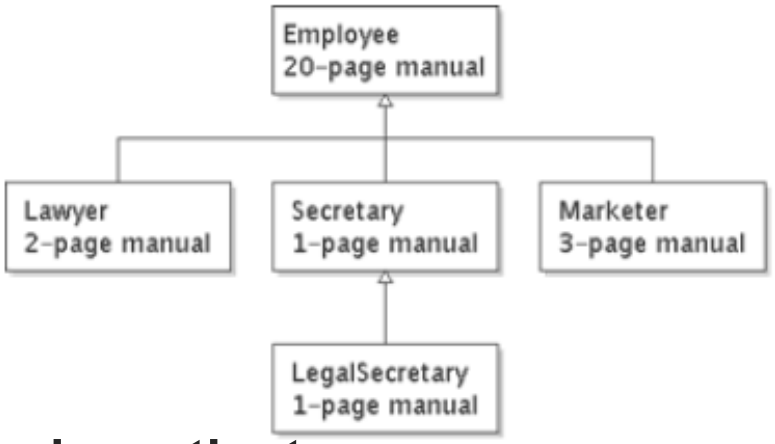
```
Employee ed = new Lawyer();
int hours = ed.getHours();   // ok; in Employee
ed.sue();                    // compiler error
```

- The compiler's reasoning is, variable `ed` could store any kind of employee, and not all kinds know how to `sue` .

To use `Lawyer` methods on `ed`, we can type-cast it.

```
Lawyer theRealEd = (Lawyer) ed;
theRealEd.sue();             // ok

((Lawyer) ed).sue();         // shorter version
```

More about casting



The code crashes if you cast an object too far down the tree.

```
Employee eric = new Secretary();
((Secretary) eric).takeDictation("hi");     // ok
((LegalSecretary) eric).fileLegalBriefs(); // error
//   (Secretary doesn't know how to file briefs)
```

You can cast only up and down the tree, not sideways.

```
Lawyer linda = new Lawyer();
((Secretary) linda).takeDictation("hi");    // error
```

Casting doesn't actually change the object's behavior.
It just gets the code to compile/run.

```
((Employee) linda).getVacationForm()        // pink
```

ArrayList and Polymorphism

```
ArrayList<BankAccount> accountList = new ArrayList<BankAccount>();
accountList.add(new BankAccount(1000));
accountList.add(new SavingAccount(2000, 2.00));
accountList.add(new CheckingAccount(10000));

// to print the balance of all accounts
for(BankAccount account: accountList) {
    System.out.println(account.getBalance());
}

// Print only the balance of Saving Account
for(BankAccount account: accountList) {
    if(account instanceof SavingAccount)
        System.out.println("saving account balance: " + account.getBalance());
}

// Get interest rate of Saving Account
for(BankAccount account: accountList) {
    if(account instanceof SavingAccount)
        System.out.println("saving account interest amount: "
                + ((SavingAccount)account).getInterestAmount());
}
```

Notes on exams

Make sure that your code compiles. Erroneous code that does not compile will suffer huge score deduction.

Use Default package. I.e., you should not have “package xxx;” declaration on top of your code.

Your code must not use third party-libraries. I.e., it should compile and run without having to install additional .jar files.

Your files will be lost if the machine reboots or crashes, so upload your current state of your submission frequently. You can keep uploading your submissions, the most recent one will be graded.